Escuela Politécnica Superior

Effective Techniques of the Use of Data Augmentation in Classification Tasks

Degree in Computer Engineering

Final Degree Project

Autor:
Unai Garay Maestre

Tutor/s:
Antonio Javier Gallego
Jorge Calvo

Universitat d'Alacant
Universidad de Alicante

June 2018

Abstract

This report follows the research and development of a final degree project of computer engineering. The purpose of this project is to accomplish a new method to overcome the lack of data. In the literature the strategy that is accustomed to achieve this task is data augmentation which is a method that artificially creates new data based on the modifications of the existing data. The heuristics underlying this modifications are very dependent on which processes are suitable for the classification task at issue. In this project we introduce an alternative using Variational Autoencoders which are powerful generative models. These are capable of extracting latent values from input variables to generate new information without the user having to take specific decisions.

Acknowledgments

First of all, I would not have been interested in this topic, Neural Networks, if it were not for the subject “Desafíos de la Programación”, taught by Juan Ramón Rico. I asked him to do this Final Project and he introduced me to my current tutors: Antonio Javier Gallego and Jorge Calvo. I am very grateful to have them. I wanted to use Neural Networks on images and they gave me this fantastic idea. They have supported me with this research all this way along, answering and helping me when I needed it. Thank you for everything.

I could not be here if it were not because of my friends. They have supported me and helped me all this way along. Thank you very much.

I also want to thank my family and friends who have supported me from the beginning until the end, in every phase of my life and with every decision I have made, and they will still do. Thank you with all my heart.

It is to them that I dedicate this work.

Intelligence is the ability
to adapt to change.

Stephen Hawking.

Contents

List of Figures

List of Tables

1 Introduction

Artificial Intelligence (AI) has become one of the most fashionable topics out there. Everybody wants to make their own contribution on this topic. It is also pure marketing, now whatever thing that has AI is better than anything (or that is what the buyer thinks). This revolution with AI has happened before but we did not have the resources to make it happen. So, what do we have currently? This time we have three important factors in our favour. Almost unlimited computing power, efficient algorithms and enormous amount of data.

Some of us confuses between Machine Learning and Artificial Intelligence. Machine Learning is a type of artificial intelligence where we no longer write rules to generate intelligence rather we create algorithm that can learn from data. In conventional programming we write a logic and give it an input, the program produces the output. In Machine learning we will give the system a set of inputs and outputs that is associated and the system will generate code for matching these inputs to the outputs. Once that is done we can use the system to produce output from another set of input.

Machine Learning itself can be classified based on the nature of learning into:

- **Supervised Learning** (Input and output is specified for training)
- **Unsupervised Learning** (Only input is given to recognise patterns)
- **Reinforcement learning** (Real world feedback is provided to system on the go)

Other classifications are also available based on the kind of output produced like classification, regression, etc.

Now, what is a neural network? Artificial neural networks are a computing system that is used for Deep Learning. Deep Learning is a type of Machine Learning which includes blocks (Function Composition) which can be adjusted on the go to produce better results. This is done by adjusting blocks that are far away from the output. Neural networks is about applying the same rules of human brain to generate intelligence. Its more about mimicking the human neurons on a silicon. Usually the blocks are arranged into multiple layers to form a deep neural network. Deep Learning is nothing but Large

Neural networks, they can be thought of as a flow chart where data comes in from one side and inference/knowledge comes out the other.

So Neural networks are fed with data. It is that data where the Neural network learns from. There is an interesting almost linear relationship in the amount of data required and the size of the model. Therefore if the complexity of the problem is high (like Image Classification) the number of parameters and the amount of data required is also very large.

1.1. Motivation

When working on a problem specific to your domain, often the amount of data needed to build models of this size is impossible to find. Just the dataset itself costs a lot of money. Sometimes you would search for a dataset on the Internet, but sometimes the dataset you are looking for just does not exist. This is where the main and biggest companies have the advantage, whether they have money and resources they could build a dataset that suits the problem they are facing.

When you do not have the money or the resources you can barely build a dataset. This dataset would not have the size to solve the problem you are dealing with if you do not want to spend that much time.

One of the solutions to face this problem could be to generate new samples from the dataset you already have. We already have a standard way to do this when it is about images. Data augmentation is well known as it generates new samples that are modified from the original one, the way you specify it.

I have made a research looking for another solution that solves it with better results. This is where generative models come out. A Variational Autoencoder (VAE) would make its work for this task. But why?

When using generative models, you could simply want to generate a random, new output, that looks similar to the training data (Unsupervised Learning), and it can be certainly done too with Autoencoders (AE). But more often, you would like to alter, or explore variations on data you already have, and not just in a random way either, but in a desired, specific direction. This is where Variational Autoencoders (VAEs) work better than any other method currently available.

1.2. Objectives

This project is all about research. When we started it there were not projects like this, focused on generating datasets with VAEs. But during this project it came out a project that was trying to replicate what we are doing here. In contrast, we compare this technology with one of the most ones used today which is data augmentation.

The objectives of this project are as follows:

- **Build an architecture for Variational Autoencoders.** We are going to explain, understand and build a main architecture that we will use and then adapt to every circumstance.
- **Generate new images with VAEs.** We will build a generative script which will generate images using the models that we saved from our Variational Autoencoders.
- **Build standard CNN with VAEs.** We will build a script that uses those images that we have just generated together with images from the original dataset using a Convolutional architecture.
- **Test both VAEs and CNN with different sizes of datasets.** Using different amounts of data we will test how the VAEs learn from that amount, how the images they produce look like and the improvement over the CNN without VAE augmentation.
- **Compare CNN vs. CNN with VAE vs. CNN with data augmentation.** Here we will compare whether VAEs improve over data augmentation or not.

2 State of the Art

2.1. Data Augmentation

The term data augmentation refers to methods for constructing iterative optimization or sampling algorithms via the introduction of unobserved data or latent variables. This can reduce overfitting of models that are fed with small datasets, because these do not generalize well from the validation and test set.

Data augmentation means increasing the number of data points. In terms of images, it may mean increasing the number of images in the dataset. In terms of traditional row/column format data, it means increasing the number of rows or objects. We will focus this technique for images.

There are many ways to augment data. In images, the original image can be rotated, change lighting conditions, crop it differently, so for one image yo can generate different sub-samples. This way the overfitting of your classifier can be reduced.

As we have seen, there are many techniques of data augmentation:

- Scaling.
- Translation.
- Rotation (at 90 degrees).
- Rotation (at finer degrees).
- Flipping.
- Adding salt and pepper noise.
- Lighting condition.
- Perspective transform.

Though the above list of image augmentation methods is not exhaustive, it comprises many widely used methods. Also, based on the use-case of the problem you are trying to solve and the type of dataset you are already having, you may use only those types of augmentations which add value to your dataset. These augmentations can be combined to produce even more number of images.

Another way to augment data is with Generative Adversarial Nets (GANs). GANs has been a powerful technique to perform unsupervised generation of new images for training. They have also proven extremely effective in many data generation tasks, such as novel paragraph generation. Furthermore, GANs have been effective even with relatively small sets of data.

In this paper [5] all these methods are explored in depth.

2.2. Improving Data Augmentation

What we have done here is a research on another methods to generate images. As we already know, data augmentation is a method that artificially creates new data based on the modifications of the existing data. The heuristics underlying this modifications are very dependent on which processes are suitable for the classification task at issue. The expectations of this project is to improve data augmentation. This is a Final Degree Project so we are limited on how we can improve over data augmentation.

To be consistent on the comparison we will be using a specific way to augment data which works very good. This method uses Variational Autoencoders. But before talking about them we have to introduce the Autoencoders. First, some history.

The idea of autoencoders was first mentioned in 1986, in an article extensively analysing back-propagation [11]. In following years, the idea resurfaced in more research papers. A 1989 paper by Baldi and Hornik helped further introduce autoencoders by offering "a precise description of the salient features of the surface attached to E (the error function) when the units are linear" [13]. Another notable paper is by Hinton and Zemel from 1994 which describes a new objective function for training autoencoders that allows them to discover non-linear, factorial representations [2]. However, it is hard to attribute the ideas about autoencoders because the literature is diverse and terminology has evolved over time. A currently emerging learning algorithm is the extreme learning machine, where the hidden node parameters are randomly generated and the output weights are computed thus learning a linear model, in a way faster than with back-propagation.

It is worth noting how all currently used variants of autoencoders have been defined in only the last 10 years:

- Vincent et al 2008 – **Denoising autoencoders**
- Goodfellow et al 2009 – **Sparse autoencoders**
- Rifai et al 2011 – **Contractive autoencoders**
- Kingma et al 2013 – **Variational autoencoders**

In addition, we should remark that autoencoders were traditionally used for dimensionality reduction and feature learning. But, as we talk about in this project, thanks to the connection between autoencoders and latent variable models they are being used as generative models.

In the next section, we will explain Autoencoders in depth and introduce Variational Autoencoders which are the ones we are going to need in this project, and why we are using them.

2.3. Autoencoders

First of all, we should talk about Autoencoders, how they work and its structure.

An autoencoder network is actually a pair of two connected networks, an encoder and a decoder. An encoder network takes in an input, and converts it into a smaller, dense representation, which the decoder network can use to convert it back to the original input.

Figure 2.3.1 is a graphic example of an encoder of a CNN.

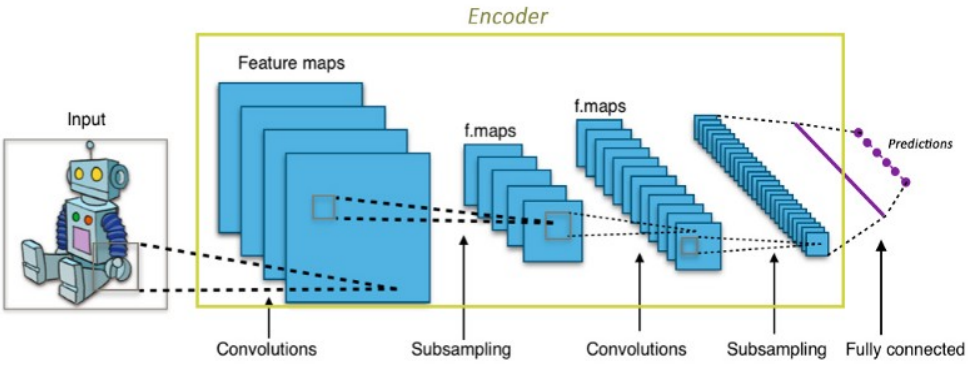


Figure 2.3.1: Encoder of a CNN

An encoder is a network that takes in an input and produces a much smaller representation (the encoding), that contains enough information for the next part of the network to process it into the desired output format. Typically, the encoder is trained together with the other parts of the network, optimized via back-propagation, to produce encodings specifically useful for the task at hand. In CNNs, the encodings produced are such that they are specifically useful for classification.

Autoencoders take this idea, and slightly flip it on its head, by making the encoder generate encodings specifically useful for reconstructing its own input.

Figure 2.3.2 is a graphic representation of an autoencoder structure:

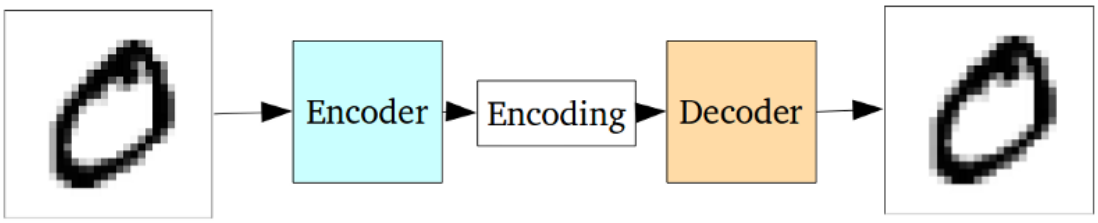


Figure 2.3.2: Autoencoder structure

The entire network is usually trained as a whole. The loss function is usually either the mean-squared error or cross-entropy between the output and the input, known as the reconstruction loss, which penalizes the network for creating outputs different from the input.

As the encoding (which is simply the output of the hidden layer in the middle) has far less units than the input, the encoder must choose to discard information. The encoder learns to preserve as much of the relevant information as possible in the limited encoding, and intelligently discard irrelevant parts. The decoder learns to take the encoding and properly reconstruct it into a full image.

2.3.1. Problem with Autoencoders

Standard Autoencoders learn to generate compact representations and reconstruct their inputs well, but asides from a few applications like denoising autoencoders, they are fairly limited.

The fundamental problem with Autoencoders, for generation, is that the latent space

they convert their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation.

For example, training an Autoencoder on the MNIST dataset, and visualizing the encodings from a 2D latent space reveals the formation of distinct clusters.

Figure 2.3.3 shows how it would look like.

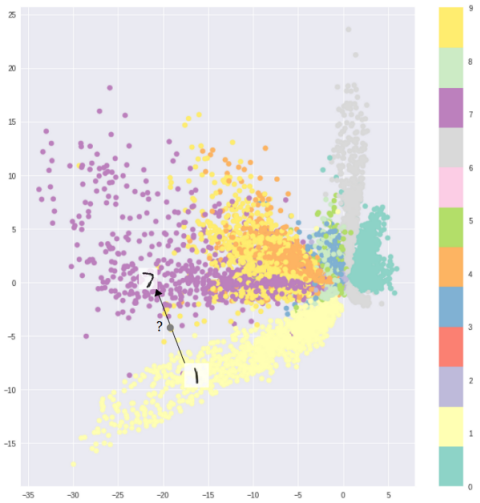


Figure 2.3.3: Autoencoder MNIST 2D latent space

But when you are building a generative model, you do not want to replicate the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

I will explain it in depth in a tutorial which starts building an Autoencoder and finishes with a generative model as Variational Autoencoders.

2.4. Variational Autoencoders

Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, by design, continuous, allowing easy random sampling and interpolation.

It achieves this by outputting two vectors of size n, instead of one: a vector of means,

μ, and another vector of standard deviations, σ.

It can be observed that in Fig. 2.4.1.

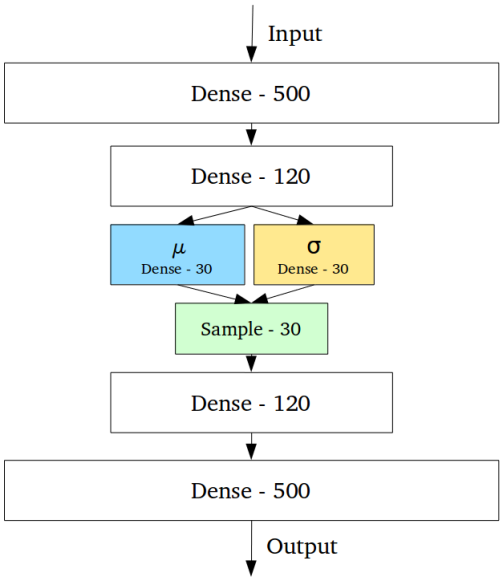


Figure 2.4.1: Variational Autoencoder structure

They form the parameters of a vector of random variables of length n, with the i th element of μ and σ being the mean and standard deviation of the i th random variable, X_i, from which we sample, to obtain the sampled encoding which we pass onward to the decoder (as seen in Fig. 2.4.2).

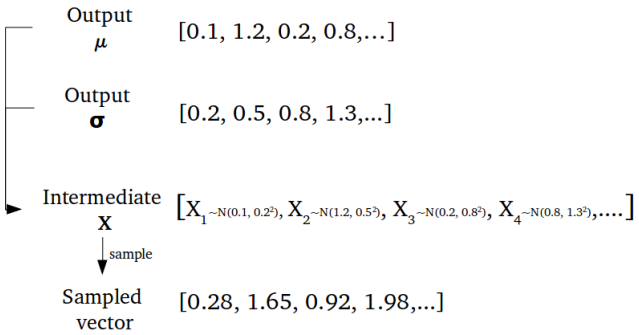


Figure 2.4.2: Sampled vector

This stochastic generation means that even for the same input, while the mean and

standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.

It can be appreciated in Fig. 2.4.3.

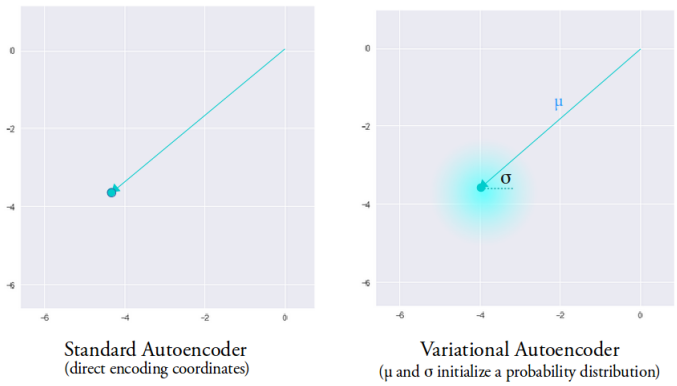


Figure 2.4.3: Encoding variation comparison

Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the "area", how much from the mean the encoding can vary. As encodings are generated at random from anywhere inside the "circle" (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well. This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the encoding of the same input during training. In code:

```
latent_size = 5
mean = Dense(latent_size)(hidden)

log_stddev = Dense(latent_size)(hidden)

def sampler(mean, log_stddev):
    # we sample from the standard normal a matrix of batch_size * latent_size
    # (taking into account minibatches)
    std_norm =
    K.random_normal(shape=(K.shape(mean)[0], latent_size), mean=0, stddev=1)
    # sampling from Z~N(μ, σ^2)
    # is the same as sampling from μ + σX, X~N(0,1)
    return mean + K.exp(log_stddev) * std_norm
```

```
latent_vector = Lambda(sampler)([mean, log_stddev])
# pass latent_vector as input to decoder layers
```

In this code above we are encoding the intermediate layer. The "hidden" layer from we start is the last layer before μ and σ as we can see in the Fig. 2.4.1. These are Dense layers which are simply layers where each unit or neuron is connected to each neuron in the next layer. We use them also in the code above to connect the last Dense layer to the two new layers reducing them to the size of the latent space we specify.

We will go in depth with the code in a tutorial further down.

The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples. Ideally, we want overlap between samples that are not very similar too, in order to interpolate between classes. However, since there are no limits on what values vectors μ and σ can take on, the encoder can learn to generate very different μ for different classes, clustering them apart, and minimize σ, making sure the encodings themselves do not vary much for the same sample (that is, less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the training data.

What we ideally want are encodings, all of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of new samples. In Fig. 2.4.4 can be seen what it would look like ideally and what we could end up with, instead.

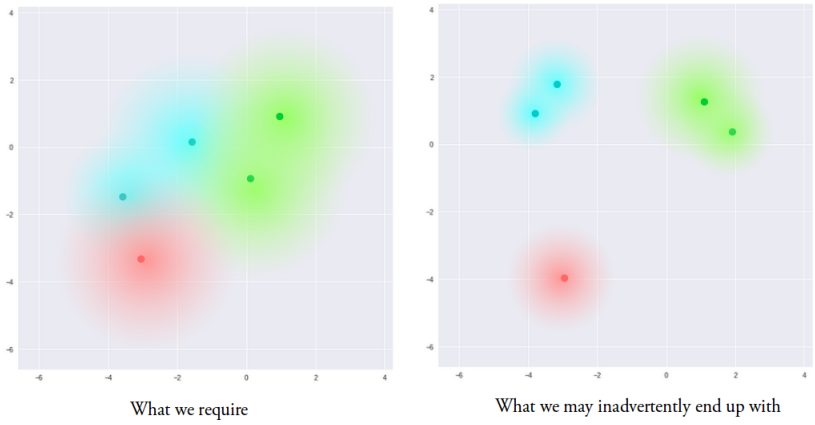


Figure 2.4.4: Ideally encoding vs. possible encoding

In order to force this, we introduce the KL divergence into the loss function. The KL

divergence between two probability distributions simply measures how much they diverge from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters (μ and σ) to closely resemble that of the target distribution. Equation 2.1 shows the KL divergence.

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1 \tag{2.1}$$

For VAEs, the KL loss is equivalent to the sum of all the KL divergences between the component X_i $N(\mu_i, \sigma_i^2)$ in X, and the standard normal. It is minimized when $\mu_i = 0$, $\sigma_i = 1$.

Now, using purely KL loss results in encodings densely placed randomly, near the center of the latent space, with little regard for similarity among nearby encodings. The decoder finds it impossible to decode anything meaningful from this space, simply because there really is not any meaning. Figure 2.4.5 shows pure KL loss results in latent space.

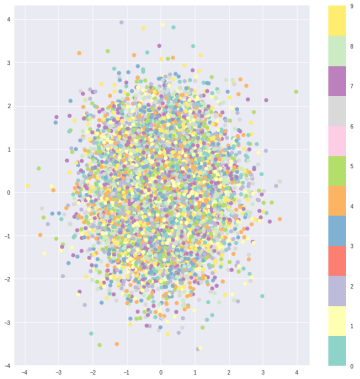


Figure 2.4.5: Pure KL loss results in 2D latent space

Optimizing the two together (reconstruction loss and KL divergence loss), however, results in the generation of a latent space which maintains the similarity of nearby encodings on the local scale via clustering, yet globally, is very densely packed near the latent space origin (compare the axes with the original). Figure 2.4.6 shows reconstruction loss and KL divergence loss results in latent space.

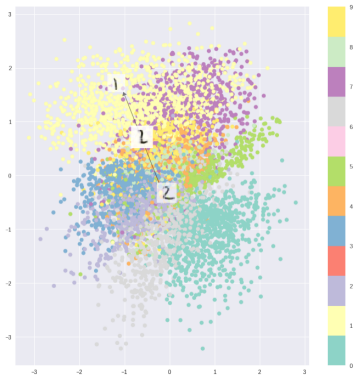


Figure 2.4.6: Reconstruction loss and KL loss results in 2D latent space

Intuitively, this is the equilibrium reached by the cluster-forming nature of the reconstruction loss, and the dense packing nature of the KL loss, forming distinct clusters the decoder can decode. This is great, as it means when randomly generating, if you sample a vector from the same prior distribution of the encoded vectors, N(0, I), the decoder will successfully decode it. And if you are interpolating, there are no sudden gaps between clusters, but a smooth mix of features a decoder can understand.

VAEs work with remarkably diverse types of data, sequential or non-sequential, continuous or discrete, even labelled or completely unlabelled, making them highly powerful generative tools.

3 Technologies

In this section we are going to talk about the technologies we have employed. When you are talking about Neural Networks it is common tu use Python as the main programming language. That would be the first tool we pretend to use.

3.1. Python

Python comes with a huge amount of inbuilt libraries. Many of the libraries are for Artificial Intelligence and Machine Learning. Some of the most used ones are Tensorflow (which is a high-level neural network library), Scikit-Learn (for data mining, data analysis and machine learning), Pylearn2 (more flexible than Scikit-Learn), etc. The list keeps going and never ends. Python has an easy implementation for OpenCV. What makes Python favourite for everyone is its powerful and easy implementation. For other languages, students and researchers need to get to know the language before getting into ML or AI with that language. This is not the case with Python. Even a programmer with very basic knowledge can easily handle Python. Apart from that, the time someone spends on writing and debugging code in Python is way less when compared to C, C++ or Java. This allows you to focus directly on what you want to do and that is exactly what the students of AI and ML want.

One of the most important libraries from Python that we are going to use is Tensorflow.

3.1.1. Tensorflow

Tensorflow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

This library is working as a backend in this project and it is another library that is running on top of it. It is called Keras.

3.1.2. Keras Library

Keras is an open source neural network library written in Python. It is capable of running on top of Tensorflow among other backends. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It does not handle itself low-level operations such as tensor products, convolutions and so on. That is why relies that work to the backend (such as Tensorflow, which does the job perfectly).

Keras contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier. The code is hosted on GitHub, and community support forums include the GitHub issues page. Moreover, it has got some datasets included such as MNIST or CIFAR10 which can be easily imported.

3.1.3. Numpy, OpenCV, Matplotlib And Others

There are many other libraries that we can import and can be very useful. As you may or may not already know is that Numpy is one of the most used libraries for Machine Learning. Numpy adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Open Source Computer Vision (OpenCV) is a common library which has C++, Python and Java interfaces and supports Windows, Linux, Mac OS, IOS and Android. OpenCV is a library of programming functions mainly aimed at real-time computer vision. OpenCV supports the deep learning frameworks TensorFlow, Torch/PyTorch and Caffe. This library has got plenty of applications for Machine Learning. We will use it to load and save data, more specifically images.

One of the most used and powerful libraries when you are visualizing data is Matplotlib. Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, Python and IPython shells, Jupyter notebooks, web application servers, and four graphical user interface toolkits. You could generate plots, histograms, power spectra, bar charts, error charts, scatterplots, etc., with just a few lines of code.

We will use some other libraries that are indispensable for our task. One of them is Random, which is a module that implements pseudo-random number generators for various distributions. Another library that we intend to use is Scikit-Learn, which is a free software machine learning library for Python. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

There are some more but these are the main ones that we will use for this task.

3.2. Jupyter Notebook

The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

- A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.

- Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

This tool is very useful when you want to test some code. It allows you to edit your code inside the browser just as easy and comfortable as your favourite code editor. It also allows you to execute code from the browser, with the results of computations attached to the code which generated them. Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library can be included in-line. There are many more of this features but the essence is that the code can be edited and executed inside the browser for testing.

3.3. Google Colaboratory

Colaboratory is a Google research project created to disseminate research and education content on machine learning. It is a Jupyter Notebook environment that requires

no configuration and runs completely in the cloud.

Colaboratory notebooks are stored in Google Drive, and can be shared as with Google Sheets or Google Docs. Colaboratory is a free service.

Colaboratory allows running Python 2 and Python 3 code, it permits to run Tensorflow and some graphs can be visualized with matplotlib among other things.

The main reason why we are going to use this tool is because you have access to a powerful graphic card to run deep neural networks in a quicker way. The graphic card that we will use it is the Nvidia Tesla K80. It allows you to run your code for twelve hours non-stop.

4 Methodology

4.1. Introduction

In this chapter we are about to describe the methodology we will follow on our project. The methodology we will adopt is based on two stages:

- **FIRST STAGE:** In this stage we are going to work with a dataset which we will use to train a Variational Autoencoder. There will be a VAE for every class on the dataset so we can train them separately. This way every VAE would learn to reconstruct every class independently and generate new samples for every class in the last step of this stage. This process can be seen in the Fig. 4.1.1.

Figure 4.1.1: First stage diagram

- **SECOND STAGE:** In this final stage we put together the original dataset and the generated samples we created in the previous stage. Subsequently, we would build an appropriate standard CNN that fits the actual problem and train it. Finally, we would make predictions with the CNN trained and observe how well it works. Figure 4.1.2 shows a diagram of this process.

Figure 4.1.2: Second stage diagram

4.2. First Stage

4.2.1. Dataset

The first step of the first stage is to choose the data we will feed to the VAEs. As we already know, the solution of generating new data is due to the problem of having a small dataset. Even though here the dataset tends to be small we have to split it in training and test set. This is because we will need to test the CNN that we will build in the second stage and the VAEs should not learn to reconstruct the test set. The test set is used to let us know how well the CNN classifies unseen data.

Before we do anything we have to split the data for every class we have in the dataset. Because we need to train a VAE for every class then we need to separate the data for every VAE. Nevertheless, the data could be already separated in classes due to the dataset structure.

It is important to have a balanced dataset. Because we are going to feed a section of the original dataset to the Variational Autoencoders, we need to ensure that there is varied data. So we will shuffle the original dataset and only then split it in training and test set.

The separation of the data could be 84% for the training set and 16% for the test set. That would be for a small dataset that some person owns. But here we are doing research and we need as many data as we can obtain to test the CNN later. If we are testing on the entire dataset then it is 84% for the training set and 16% for the test set. But if we are reducing the training set to see how the VAEs are learning we should not decrease the test set because the results would not be consistent. So the test set would maintain the percentage over the entire dataset. With a graphic circle 4.2.1 represents train and test set percentage.

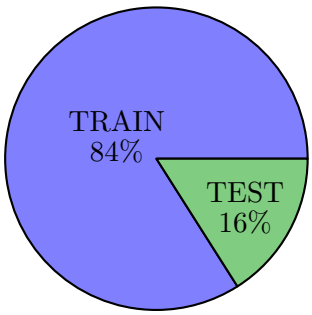


Figure 4.2.1: Train test percentage

This percentage generally gives good results, but depending on the problem that you are facing you should choose your own percentage.

Diagram 4.2.2 sums up all the actions that we have done so far inside the Dataset step.

Figure 4.2.2: Dataset steps

Now, the training set that we have obtained by splitting the data is the one that we feed to the Variational Autoencoders. In the next step we explain the training of the VAEs using this data.

4.2.2. Train Variational Autoencoder

In this step of the first stage we build a Variational Autoencoder for every class of the dataset. We have already seen how Variational Autoencoders work so we would just have to build it in Python as we will see in the implementation section.

After that we have to feed every VAE with the data that we have prepared for every class in the last step. As we already know, we have to split the data in classes, shuffle every class and split it in training and test set.

Once we have the training and test set with their respective labels we start to train every VAE. Then, we plot the results in the latent space to see how well the Variational Autoencoder has learned to reconstruct the data. Moreover, we can plot some examples of the new data generated building a generator from which we will benefit later. This way we can see how the generated samples will be.

To end this second step of the first stage, we save the model in a folder. There will be a folder for every class with a model in it. We save the VAE model, the encoded model and the generator model.

4.2.3. Generate New Samples

In this last step of the first stage, we finally generate the samples from our Variational Autoencoders. We have already use a generator to get an idea of what we were going to generate as new data. So we are about to adopt the same structure model that we have applied in the VAEs to do that. In order to do it, we have to load the models that we have saved for every class.

Once we have loaded the models, we are ready to start generating new samples. The samples generated are saved in another folder that we will use later. We need to generate new samples, but we can not do it without changing the encoded part, because otherwise we would be getting samples too similar to the original ones, and that would not increase the accuracy of the CNN that we would build later.

As we observe in Fig. 4.2.3, you could reconstruct the same images if you put the same input you put in the Encoder. That is, the Z value, which is a vector. We could generate a random vector between two values giving it the size of the latent space we are dealing with and feed that to the generator (the decoder) and predict new samples.

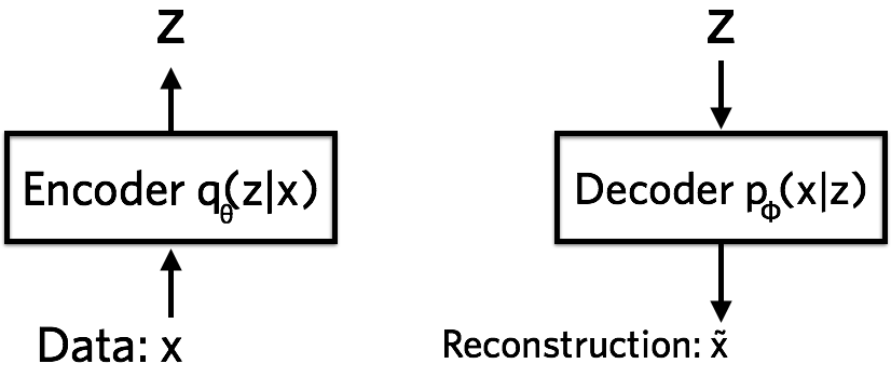


Figure 4.2.3: Encoder decoder reconstruction

The goal is to generate enough data that a proper result can be obtained with the neural network you build later. So, for instance, if you had 500 samples of the original data you could generate 20000 more to get better results. It depends on the type of data you are dealing with.

Now we move to the second and final stage.

4.3. Second Stage

4.3.1. Dataset + Generated Samples

In this step of the second stage, we get together the original dataset and the generated samples we just created in the last step. As we said earlier on, the original dataset was separated in training and test set, and the training set was what we fed to the Variational Autoencoders to train them. The test set remains intact and we would use it now.

The training set of the original data and the generated samples come together to form the training set for the new neural network and the test set is the one we utilize to get to know how the NN learns.

Figure 4.3.1 explains how the union of the data works.

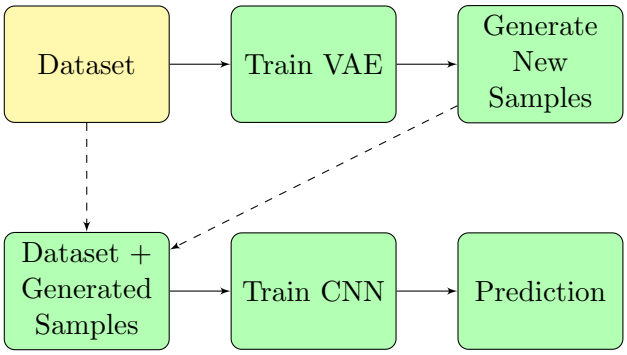


Figure 4.3.1: Data augmented workflow

After preprocessing the data we go to the next step which is to train the neural network.

4.3.2. Train CNN

In this second step of the second stage, we build a Convolutional Neural Network. This CNN is trained with the original dataset to see how well the CNN is doing with that data.

Subsequently, we feed the CNN with the dataset adding the new samples that we generated and compare it with the CNN trained with just the original dataset.

We also train the CNN using data augmentation to see if our methodology is improving over it.

4.3.3. Prediction

In the last step of the second and last stage of our methodology, we predict values with every CNN we have trained. Moreover, we compare the results and analyze them to see which method has obtained the best results. Finally, we do the same process comparing them with different sizes of the dataset to see when the methods are better than the others in different situations.

The figure 4.3.2 shows a summary of the entire process.

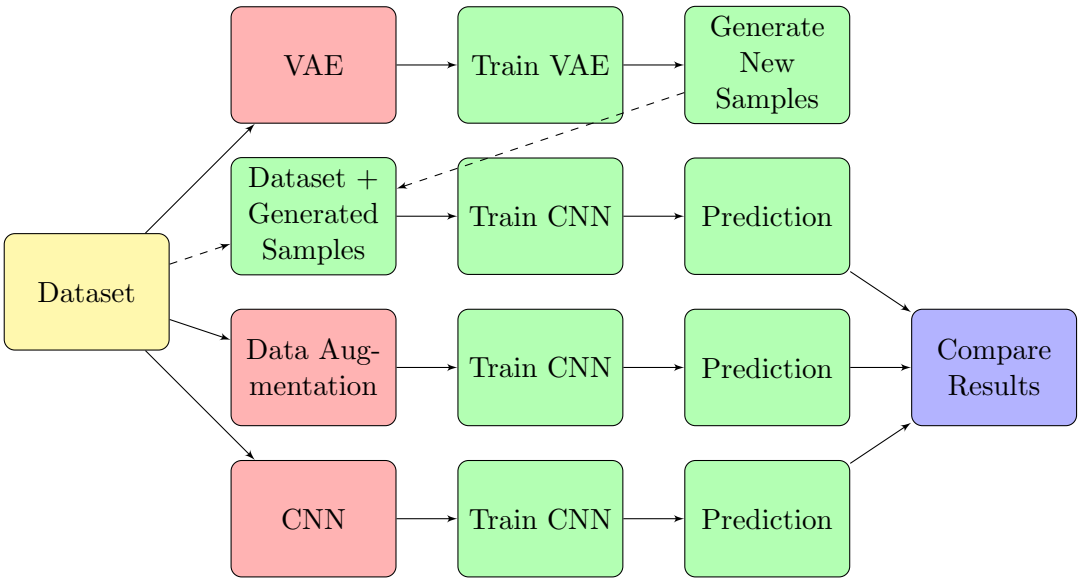


Figure 4.3.2: Methodology summary

As we observe in this figure, there are three processes and in the end we compare them to see which one is the best and which one is the worst. We do this process with different sizes of the dataset. So it would a loop within every iteration, the only thing that changes is the size of the original dataset. In case of the Data Augmentation step, the extra data is implicit in the "Train CN" process.

As we already know the methodology that we intend to follow, the implementation of it can begin.

5 Implementation

5.1. Introduction

In this chapter we dig deeper into the implementation. We will start from the bottom. We have already talked about Autoencoders, which are the basis to understand Variational Autoencoders, our main goal. Here we will see how we can go from a simple Autoencoder to a complex Variational Autoencoder. But this time we will go deeper and show how we can implement each step in Python. Furthermore, we will implement a CNN to make the experiments later on. This section is intended to follow these steps:

- **Autoencoder and Variational Autoencoder**
  - **Simplest Possible Autoencoder:** A basic Autoencoder to understand the basis.
  - **Variational Autoencoder:** A basic implementation of a Variational Autoencoder.
  - **Convolutional Variational Autoencoder:** A more complex VAE with convolutions.
- **Convolutional Neural Network:** Build a general CNN to make experiments with different techniques and configurations.

5.2. Autoencoder and Variational Autoencoder

5.2.1. Simplest Possible Autoencoder

We start with a simple example of autoencoder which consists of a single fully-connected neural layer as encoder and as decoder. As we have seen before, an encoder is a network that takes in an input and produces a much smaller representation (the encoding), that contains enough information for the next part of the network to process it into the desired output format. It learns to preserve as much of the relevant information as possible in the limited encoding, and intelligently discard irrelevant parts. On the other hand, we have the decoder, which learns to take the encoding and properly reconstruct it trying to map the original input. We will be working with images so the input would be an image that passes through the encoder and then the decoder would try to reconstruct it completely.

First, we declare two variables as the encoded and decoded part. The encoded takes the original image as input, which could be a vector of 784 dimensions (input_img), and encode it to "encoding_dim", which is set at 32. The decoded takes the encoded part as input and tries to reconstruct the original image giving an output of the same size as the image that was fed in the beginning, which in this case could be 784.

```
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
```

Afterwords, we declare a model that maps an input image to its reconstruction (decoded representation), which we call "autoencoder". We also declare another variable called encoder, which overwrites the last variable, as a model that maps the original image to its encoded representation.

```
autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
```

To get finished with the models, we declare a decoder model which takes an "encoded_input" (a placeholder for an encoded input) and maps it with a "decoder_layer", which is the autoencoder model without the last layer, that takes the "encoded_input" as input.

```
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

We compile the "autoencoder" model with "adadelta" [1] as the optimizer, which works fine, and "binary_crossentropy" as loss function.

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

If we wanted to predict a result to see how the autoencoder reconstructs the input, we would use the predict() function with the test set on the encoder, which gives us encoded images that we feed to the predict() function of the decoder:

```
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

This would be the implementation of a basic autoencoder which fits perfectly with which we have already explained in other chapters. But we need to reconstruct the data with a variation from the original input, so we need Variational Autoencoders. Let us talk about the implementation of a basic Variational Autoencoder.

5.2.2. Variational Autoencoder

We have already talked about Variational Autoencoders. The main property that separates them from the Autoencoders is their latent spaces. These are continuous, allowing easy random sampling and interpolation. For what we need, which is a generative model that suits our main goal in this project, they work perfectly.

We have already talked about how they work mathematically and functionally. Here we are about to show how they are implemented.

First, we map the inputs to our latent distribution parameters:

```
x = Input(shape=(original_dim,))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim)(h)
```

The term "original_dim" is related with the size of the input which, for instance, could be 784. It has the same meaning as the term "input_img" that we explained above. The concept "intermediate_dim" is the size of the encoded part and "latent_dim" is the latent dimension size.

We can deduce that "z_mean" is related to μ and "z_log_sigma" to σ.

We use these parameters to sample new similar points from the latent space:

```
def sampling(args):
```

```
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(batch_size, latent_dim),
                              mean=0., std=epsilon_std)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_sigma])
```

The "Lambda" function is a layer from provided by Keras that wraps an arbitrary expression as a Layer object. Using the method "sampling" and Lambda we are obtaining the sampled encoding from the mean and standard deviation. This sampled encoding then it will be passed onward to the decoder.

This last process we have just completed is the one that we already explained in the introduction. Figure 5.2.1 shows the already explained structure that this process follows.

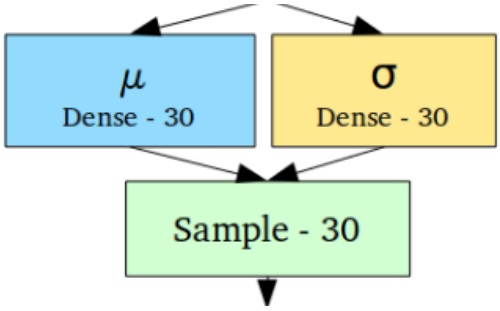


Figure 5.2.1: VAE encoded structure

What we have done so far allows us to instantiate 3 models:

- An end-to-end autoencoder mapping inputs to reconstructions.
- An encoder mapping inputs to the latent space.
- A generator that can take points on the latent space and will output the corresponding reconstructed samples.

This models are declared in the following code:

```
vae = Model(x, x_decoded_mean)

encoder = Model(x, z_mean)
```

```
# Generator
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)
decoder_input = Input(shape=(latent_dim,))
_h_decoded = decoder_h(decoder_input)
_x_decoded_mean = decoder_mean(_h_decoded)
generator = Model(decoder_input, _x_decoded_mean)
```

We train the model using the end-to-end model, with a custom loss function: the sum of a reconstruction term and the KL divergence regularization term.

```
def vae_loss(x, x_decoded_mean):
    xent_loss = objectives.binary_crossentropy(x, x_decoded_mean)
    kl_loss = - 0.5 * K.mean(1
              + z_log_sigma
              - K.square(z_mean)
              - K.exp(z_log_sigma), axis=-1)
    return xent_loss + kl_loss

vae.compile(optimizer='rmsprop', loss=vae_loss)
```

We include the function that we have just created as the loss function of the model, applying a different optimizer.

As Dense layers are not enough if we are dealing, for instance, with images or more complex inputs, we need to get a deeper and stronger model. That is why we are about to build a Convolutional Variational Autoencoder.

5.2.3. Variational Autoencoder with Convolutions

Convolutional Neural Networks (CNN) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. CNNs derive their name from the "convolution" operator. The primary purpose of Convolution in case of a CNN is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.

In Convolutional Neural Networks, you use a matrix called filter or kernel that you slide over the input image and compute the dot product. These filters act as feature

detectors from the original input image. In practice, a CNN learns the values of these filters on its own during the training process. The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

So instead of using Dense layers we use Convolutional layers.

Below is shown the changes that we would have to do on the encoder.

```
x = Input(shape=original_img_size)
conv_1 = Conv2D(img_chns,
                kernel_size=(2, 2),
                padding='same', activation='relu')(x)
conv_2 = Conv2D(filters,
                kernel_size=(2, 2),
                padding='same', activation='relu',
                strides=(2, 2))(conv_1)
conv_3 = Conv2D(filters,
                kernel_size=num_conv,
                padding='same', activation='relu',
                strides=1)(conv_2)
conv_4 = Conv2D(filters,
                kernel_size=num_conv,
                padding='same', activation='relu',
                strides=1)(conv_3)
flat = Flatten()(conv_4)
hidden = Dense(intermediate_dim, activation='relu')(flat)

z_mean = Dense(latent_dim)(hidden)
z_log_var = Dense(latent_dim)(hidden)
```

As for the decoder, we will need Convolutional layers too, but this time they will be called Conv2dTranspose, also called Deconvolution, used to decode the input of the decoder to map the input of the encoder.

```
decoder_hid = Dense(intermediate_dim, activation='relu')
decoder_upsample = Dense(filters * 14 * 14, activation='relu')

if K.image_data_format() == 'channels_first':
    output_shape = (batch_size, filters, 14, 14)
else:
    output_shape = (batch_size, 14, 14, filters)
```

```
decoder_reshape = Reshape(output_shape[1:])
decoder_deconv_1 = Conv2DTranspose(filters,
                                   kernel_size=num_conv,
                                   padding='same',
                                   strides=1,
                                   activation='relu')
decoder_deconv_2 = Conv2DTranspose(filters,
                                   kernel_size=num_conv,
                                   padding='same',
                                   strides=1,
                                   activation='relu')
if K.image_data_format() == 'channels_first':
    output_shape = (batch_size, filters, 29, 29)
else:
    output_shape = (batch_size, 29, 29, filters)
decoder_deconv_3_upsamp = Conv2DTranspose(filters,
                                          kernel_size=(3, 3),
                                          strides=(2, 2),
                                          padding='valid',
                                          activation='relu')
decoder_mean_squash = Conv2D(img_chns,
                             kernel_size=2,
                             padding='valid',
                             activation='sigmoid')

hid_decoded = decoder_hid(z)
up_decoded = decoder_upsample(hid_decoded)
reshape_decoded = decoder_reshape(up_decoded)
deconv_1_decoded = decoder_deconv_1(reshape_decoded)
deconv_2_decoded = decoder_deconv_2(deconv_1_decoded)
x_decoded_relu = decoder_deconv_3_upsamp(deconv_2_decoded)
x_decoded_mean_squash = decoder_mean_squash(x_decoded_relu)
```

In the penultimate layer "decoder_deconv_3_upsamp" utilizes a bigger size of "strides" to increment the size of the image as is just one layer away to output the reconstructed image.

The rest is the same, we just needed to get a better model for the encoder and the decoder.

5.3. Convolutional Neural Network

In this section we pretend to build a general Convolutional Neural Network which is the one that would be used to train the original problem. We want to adapt it to be executed for the original problem without any addition of data on the original dataset, for the data composed of the original dataset and the generated samples of the Variational Autoencoders, and for other techniques of data augmentation.

The technique we are going to adopt for data augmentation it is called the Keras Image Augmentation API.

5.3.1. Keras Image Augmentation API

Like the rest of Keras, the image augmentation API is simple and powerful.

Keras provides the ImageDataGenerator class that defines the configuration for image data preparation and augmentation. This includes capabilities such as:

- Sample-wise standardization.
- Feature-wise standardization.
- ZCA whitening.
- Random rotation, shifts, shear and flips.
- Dimension reordering.
- Save augmented images to disk.

We use the ImageDataGenerator class as shown in the code down below:

```
datagen = ImageDataGenerator(
            rotation_range=20,
            width_shift_range=0.2,
            height_shift_range=0.2,
            horizontal_flip=False
          )
```

Then we fit the model using the fit_generator() function instead of fit():

```
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=args.b),
                    steps_per_epoch=len(X_train),
                    validation_data=(X_test, Y_test),
                    epochs=args.e,
                    verbose=1 if args.v == True else 2,
                    callbacks=[early_stopping, checkpoint])
```

This function calls the data augmentation method of Keras in each iteration, which generates random augmented samples with the configuration that has been indicated from the input data.

5.3.2. Model

In this section it is described the CNN model employed to evaluate the different techniques explained for data augmentation.

The model of the CNN would be the same for the three implementations. What changes is the pre-process of the data depending on whether you get just the original dataset or you add the generated samples, and the fitting of the model.

In Python, the model looks like this:

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
        padding='same', input_shape=input_shape))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

This is a strong model that would do fine with a small dataset and a big dataset such

the one generated by the VAEs.

5.3.3. Parameters

To use the code in a single file and be able to execute the three techniques we need to parametrize it:

```
parser = argparse.ArgumentParser(description='CNN')
parser.add_argument('-path', required=True, help='Dataset path')
parser.add_argument('-vae', default=None, help='Path to VAE augmentation folder')
parser.add_argument('--aug', action='store_true', help='Use Keras augmentation')
parser.add_argument('-limtr', default=100, help='Limit train size. -1 to load all i
parser.add_argument('-limte', default=125, help='Limit test size. -1 to load all im
parser.add_argument('-e', default=100, help='nb epochs')
parser.add_argument('-b', default=32, help='batch size')
parser.add_argument('--v', action='store_true', help='Activate verbose')
args = parser.parse_args()
```

As shown above, there is a parameter to introduce the samples produced by the VAEs, there is a parameter to enable Keras data augmentation, and without including any of them you would be using the CNN in the original way.

In the next chapter we will be doing experimentations with the techniques we have just mentioned to see if our method is worth it.

6 Experimentation

In this chapter we are about to test our method comparing it with the standard way to do data augmentation.

To test it out we are going to benefit from a common dataset that has proven to be a good way to teach the basis of Machine Learning, but also to be a simple way to compare different methods due to its simplicity. We will bring into play the MNIST handwritten digit dataset.

6.1. MNIST dataset

The MNIST dataset (Modified National Institute of Standards and Technology dataset) is a large dataset of handwritten digits that is commonly used for training image processing systems. It was created by "re-mixin" the samples from NIST's original dataset (in this paper [14] there is more information about NIST's dataset).

The Fig. 6.1.1 shows four examples of this dataset.

Figure 6.1.1: MNIST's handwritten digits

It also includes labels for each image, telling us which digit it is. For example, the labels for the images of Figure 6.1.1 are 5, 0, 4, and 1.

The goal of a model for this dataset would be from an input image to predict what digit it is.

The MNIST data is split into two parts: 60,000 data points of training data and 10,000

points of test data.

As mentioned earlier, every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. Both the training set and test set contain images and their corresponding labels.

Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers as observed in the Fig. 6.1.2.

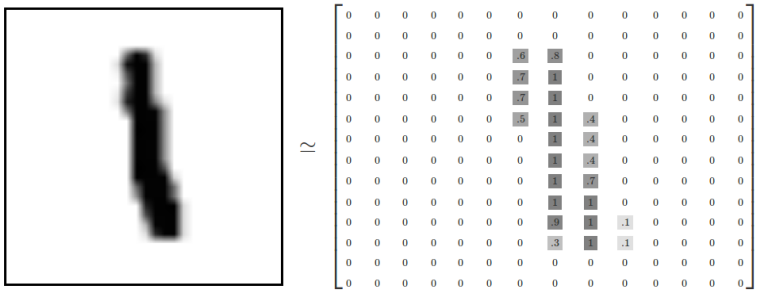


Figure 6.1.2: MNIST digit as a matrix

So we have 60,000 images of training that we can represent as matrices and 10,000 images of test that we can also represent as matrices.

But we do not want the entire dataset, we want a small amount of it. We will be testing out the methods with different sizes of the original dataset.

Figure 6.1.3 shows the directory structure of the dataset.

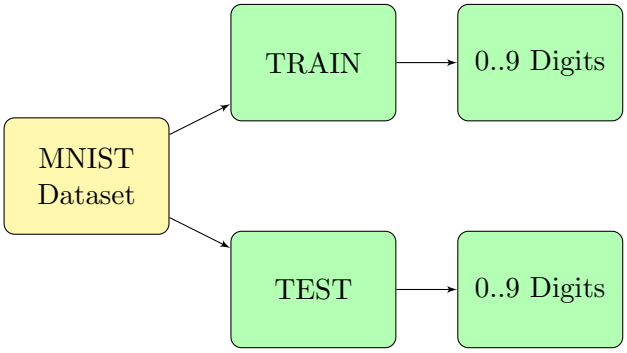


Figure 6.1.3: MNIST's directory structure

To train the VAEs we will be using just the "TRAIN" folder separating it again into train and test. We will be using both folders to train and evaluate the CNN.

For this dataset we will be using a specific method to load the data. It is the same method for the VAEs and the CNN but the output vectors of the function would be different as we are using different models to train the data. This method for the VAEs would look like this:

```
def load_images(path):
    X = []
    Y = []
    label = str(digit)
    for fname in list_files( path, ext='png' ):
        img = cv2.imread(fname, cv2.IMREAD_UNCHANGED)
        X.append(img)
        Y.append(label)

    combined = list(zip(X, Y))
    random.shuffle(combined)
    X[:], Y[:] = zip(*combined)

    X = X[:TOTAL_IMAGES]
    Y = Y[:TOTAL_IMAGES]

    X = np.asarray(X).astype('float32')
    X = X/255.

    Y = np.asarray(Y)

    return X, Y
```

First, we are loading the entire training dataset. Then we shuffle it to get varied data but using a specific seed to get the same variety every time we shuffle the data at the beginning. After that, we get the number of images we want to load by using the variable "TOTAL_IMAGES". Finally, we normalize the data and return it as output.

Moreover, we split the data for training and test, as we also need a test set. The percentage will be 84% for the training set and 16% for the test set.

6.2. Train VAEs

Once we have loaded the data we can start training our VAEs. Before starting the process of training every VAE we are going to show a couple of visualizations that a VAE can make. For this, we train a VAE for all the digits and use the entire dataset.

As we are training it for all of them, this VAE would be trained to understand how all the digits vary one from each other.

Because we are dealing with a latent space of two dimensions, we could look at the neighbourhoods of the different classes on a 2D plane. See Fig 6.2.1.

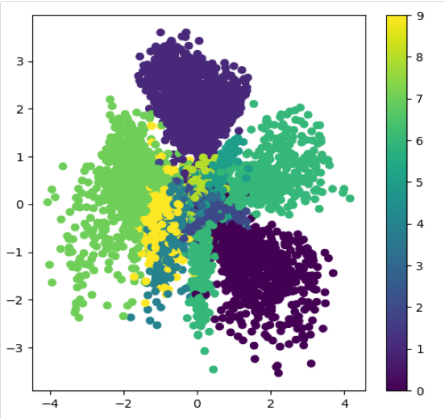


Figure 6.2.1: Convolutional VAE using MNIST and a 2D latent space

Each of the coloured clusters in Fig. 6.2.1 is a type of digit. Close clusters are digits that are structurally similar.

As we are also building a generator, we can generate new images using it. We can change how the digits look like from the original dataset just varying the encoded part. Because we are training a VAE for all of them we can not choose what digit we generate (see Fig. 6.2.2). That is why we need to implement one VAE for every digit. Using one VAE for every digit we can generate variations of the same kind of digit.

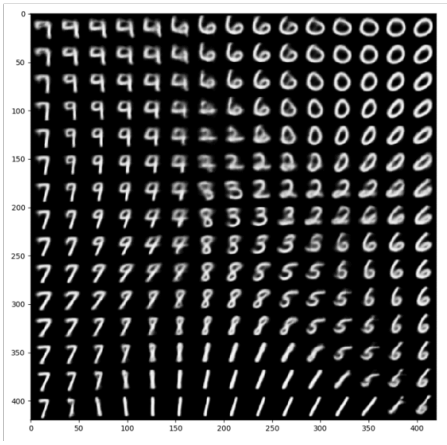


Figure 6.2.2: Convolutional VAE using MNIST digits

6.2.1. Configuration of VAEs

As we will be dealing with small amounts of data, we need to choose which configuration we are going to adopt to train every VAE for certain sizes. To do that, we run some tests on the VAE for the digit 0 which will be giving us an idea on which configuration we should use for all the digits.

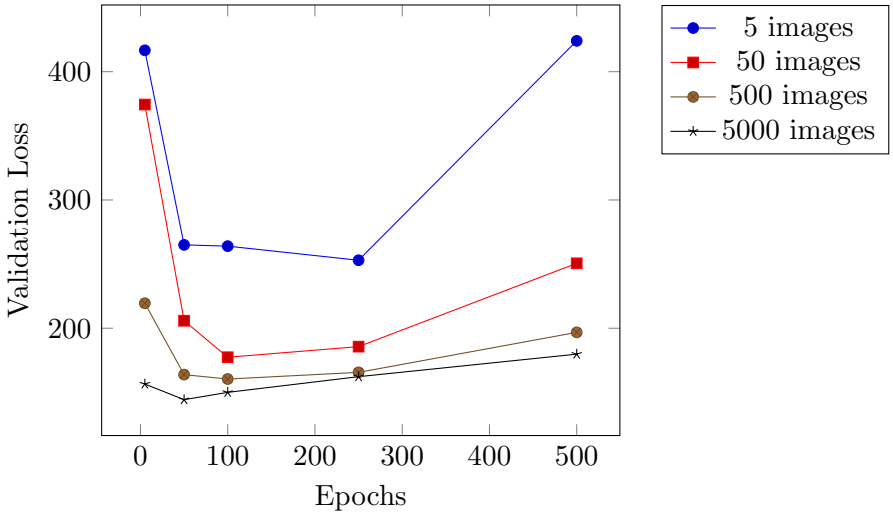
For the tests we will be using different number of epochs: 5, 50, 100, 200 and 500 and different sizes.

The results of the test look like the ones in table 6.1, that is represented graphically in Fig. 6.2.3.

Table 6.1: VAE validation loss for digit 0

	5 epochs	**50 epochs**	**100 epochs**	**250 epochs**	**500 epochs**
5 images	416,68	265	264	253,01	424,05
50 images	374,40	205,85	177,38	185,66	250,60
500 images	219,50	163,83	160,38	165,60	196,78
5000 images	156,51	144,31	150,00	162,24	179,79

Figure 6.2.3: Graph validation loss VAE digit 0



As we can observe, as less images we have higher is the validation loss of the VAE, which means it is making bigger mistakes when reconstructing the digits.

We can also interpret that as many epochs you do less validation loss you get. This is

usually true except when you have too many that you overfit the model and it starts increasing the validation loss, the model fits too much to seen data and does not generalize well.

This could be fixed by adding a callback to the model. This could be the situation to use Early Stopping, which what does is to stop the training of the model when the validation loss increases in a specific number of epochs (patience).

The problem here is that the validation loss is that volatile that it can not be told whether the model will improve or not. Either you could add a big "patience" to the callback to not overfit the model independently of the number of images you are getting as input or depending on the amount of images you could increase or decrease the epochs.

Generally, the sweet spot would be between 50 epochs and 200 epochs, depending on the amount of data. As we can see in the Fig. 6.2.3, for high amounts you should use less epochs as shown in the black line between 50 and 100 epochs where the validation loss increases. Also for small amounts you should use more epochs as we can see in the blue line between 100 and 250 epochs where de validation loss decreases.

Overall, we will be using small amounts of images so we will use more than 100 epochs. We will not use the callback, as we want to see for every size of the dataset how many epochs we need to achieve the sweet spot.

6.3. Tests

Now we are about to run some varied tests to see if VAEs improve over data augmentation and a normal CNN without any enhancement. The structure of this section is as follows:

- **50 images:** train with 4 images for each digit and validate with 1.
- **100 images:** train with 8 images for each digit and validate with 2.
- **250 images:** train with 21 images for each digit and validate with 4.
- **500 images:** train with 42 images for each digit and validate with 8.
- **1000 images:** train with 84 images for each digit and validate with 16.

In every one of them we will put the methodology into practice which, as a reminder, was to train the corresponding VAEs, generate new images, put together the original

dataset and the generated images and the train the CNN. At the same time, we are running the same test to train the CNN without any enhancements and with data augmentation. Subsequently, we compare the results.

6.3.1. 50 Images

First of all, we have to train the VAEs. As 5 images for each VAE is a incredibly small amount of data so the VAE struggles to learn how to reconstruct those digits, taking into account that 1 of those digits is for testing (84% training and 16% testing), the digits generated are not going to look that well.

In this test we are about to see whether some digits that look not quite like the originals can help to improve the result of the original CNN.

Training VAEs and generating new images

First we train the VAEs using 5 images for each digit. For the configuration of the VAEs we will be using 200 epochs as for small amounts of data it works well.

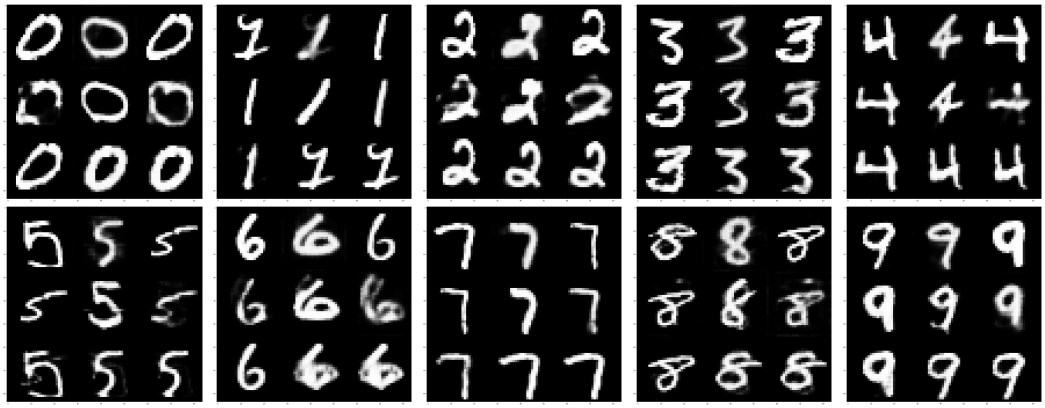


Figure 6.3.1: Generated digits using 50 images and a 2D latent space

As we can see, the digits do not look that well but for 4 digits of training and 1 for testing for every digit they have been reconstructed quite well. We would say that the VAE is working really well even for that small amount of data. We can also see that the digit 5 is struggling to reconstruct itself introducing a bit of noise which can be bad for the CNN.

These are the digits that are shown as a example of generating new images from the VAEs, but we have not generated new ones yet.

We run the generator which makes use of the same structure as the VAE but now we will be saving those images in a folder using OpenCV so we can use them later on the CNN.

Now the question is, how many samples should we generate to increase the accuracy of the CNN without overfitting it? We try with the amount of 2500 for every digit. This is 25000 more samples in total, which if you add them to the original dataset they sum 25050.

It is time to see the results of the method that we have used compared to the usual method and data augmentation.

First we train the CNN without any enhancements, using the usual method.

CNN with no extra images

Before we show the results, we have to explain some technicalities about them. Apart from showing the loss and accuracy of the test set, we are about to show another concepts like precision, recall, f-measure (f1-score) and support.

The precision is the ratio tp / (tp + fp) where tp is the number of true positives (digits correctly classified) and fp the number of false positives (samples of other classes incorrectly classified). The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives (samples of the positive class incorrectly classified as negatives). The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and its worst score at 0.

The F-beta score weights recall and precision by a factor of beta. beta == 1.0 means recall and precision are equally important.

The support is the number of occurrences of each class.

These are metrics of binary classification, where a class is evaluated (positive class) against the rest (negative classes). These metrics are calculated for all the classes and, ultimately, the mean result is returned.

Now, let us start with the table 6.2 where it is shown how well the neural network has classified every digit based on the precision, recall, f1-score and support.

Table 6.2: CNN with no extra images using 50 images

Digit	**precision**	**recall**	**f1-score**	**support**
0	0.71	0.82	0.76	700
1	0.92	0.90	0.91	700
2	0.73	0.82	0.77	700
3	0.61	0.77	0.68	700
4	0.91	0.73	0.81	700
5	0.80	0.72	0.76	700
6	0.94	0.72	0.81	700
7	0.83	0.70	0.76	700
8	0.66	0.60	0.63	700
9	0.66	0.83	0,74	700

The number that the CNN has predicted better is the 1 as it is seen, the f1-score gives us a better and consistent understanding of which digit has predicted better. The support is always 700 because for each digit we have provided 7000 samples for testing in total.

Now we will be giving the average results for the training of the CNN with 50 images in the following table 6.3.

Table 6.3: CNN average results of 50 images

Loss	0.7904
Acc	0.760
Precision	0.7771
Recall	0.7610
F1	0.7630
Support	7000

Overall, in the next tests we will be looking to the loss, the accuracy and the f1-score as the parameters that will decide which method is better in every situation.

As for this test, we could say that using just 50 images it is getting an accuracy of 76%

which is very good for the small data that we are using. This is due to the simplicity of the images and the model that we are using.

CNN with VAE generated images

Now we are going to show the results of the CNN using the 25000 samples that we generated with the VAEs we trained.

Table 6.4: CNN VAE using 50 images and a 2D latent space

Digit	**precision**	**recall**	**f1-score**	**support**
0	0.92	0.81	0.86	700
1	0.77	0.99	0.86	700
2	0.79	0.69	0.74	700
3	0.81	0.85	0.83	700
4	0.79	0.80	0.79	700
5	0.94	0.86	0.90	700
6	0.82	0.90	0.86	700
7	0.93	0.61	0.73	700
8	0.66	0.77	0.71	700
9	0.73	0.78	0.76	700

The digit that is being predicted better is the 5. The 1 was predicted better with the normal CNN but overall this model looks to be doing better.

The average results are shown in the next table 6.5.

Table 6.5: CNN VAE average results 50 images 2D latent space

Loss	1.8515
Acc	0.8049
Precision	0.8155
Recall	0.8049
F1	0.8037
Support	7000

As for the loss it shows clearly that is doing much worse than the CNN but it is due to the amount of data that we fed to the neural network and because there are some of the samples that give extra noise. This also tells us that it could be overfitting. As for the accuracy it shows that is doing better than the CNN itself.

CNN with data augmentation

Now we will see the results of the CNN with the data augmentation process of Keras in table 6.6.

Table 6.6: CNN Keras data augmentation using 50 images

Digit	precision	recall	f1-score	support
0	0.93	0.85	0.89	700
1	0.96	0.99	0.98	700
2	0.95	0.86	0.90	700
3	0.85	0.90	0.88	700
4	0.91	0.96	0.93	700
5	0.96	0.95	0.95	700
6	0.83	0.92	0.87	700
7	0.80	0.60	0.69	700
8	0.84	0.70	0.76	700
9	0.66	0.90	0.76	700

The predictions are much better than the previous methods. It is predicting better the digit 5, like the CNN with VAE, but it seems to be doing better overall.

Now let us take a look to the average results:

Table 6.7: CNN with data augmentation average results using 50 images

Loss	0.4608
Acc	0.8621
Precision	0.8690
Recall	0.8621
F1	0.8610
Support	7000

The loss is lower than the one of the CNN itself and much lower than the CNN with VAE. The accuracy is higher.

Summary

To end this 50 images test, we will compare side by side the average results of every method in table 6.8.

Table 6.8: Summary of average results different methods using 50 images and a 2D latent space

	CNN	VAE	AUG
Loss	0.7667	1.8515	0.4608
Acc	0.7611	0.8049	0.8621
Precision	0.7785	0.8155	0.8690
Recall	0.7611	0.8049	0.8621
F1	0.7626	0.8037	0.8610
Support	7000	7000	7000

As we have already said, the loss of the data augmentation method is lower than any of the other methods which tells us is giving less errors on the validation set. The accuracy of the CNN with VAE is higher than the CNN itself by 4% of difference but lower than the data augmentation by 6% of difference. The f1-score tells us the same as the accuracy in this case.

The conclusion for this test is that with a really small amount of data our method does much better than the CNN itself but it is far away from the current used method that is data augmentation.

6.3.2. 100 Images

Now we will be testing 100 images in total, which is 10 images per digit.

Training VAEs and generating new images

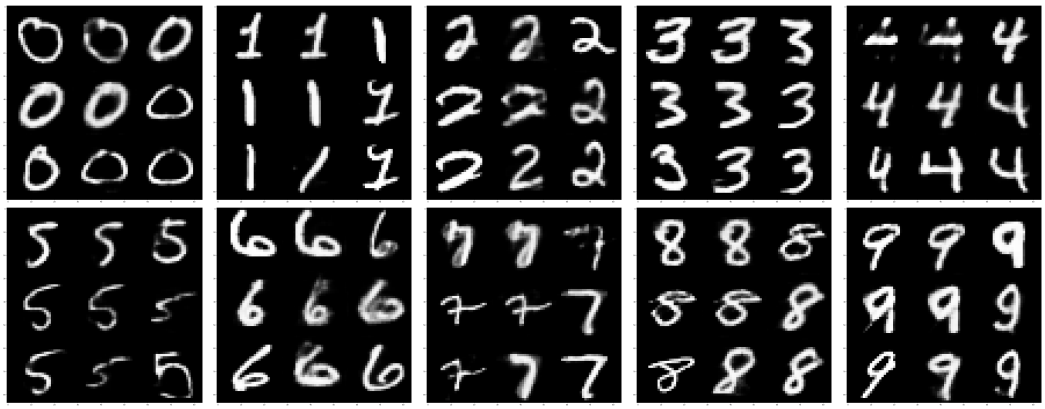


Figure 6.3.2: Generated digits using 100 images and a 2D latent space

Some of the digits that we have generated look better than the last test but some are not well reconstructed. Adding new images it does not necessarily mean that they are going to be reconstructed better but also that there will be a bigger variety of digits, meaning how differently they are written.

In this case we are doubling the amount of data from the last test, let us see how the methods learn and predict from 100 images.

CNN with no extra images

Table 6.9 shows the results of the CNN trained with 100 images.

Table 6.9: CNN with no extra images using 100 images

Digit	**precision**	**recall**	**f1-score**	**support**
0	0.90	0.89	0.90	700
1	0.97	0.91	0.94	700
2	0.76	0.87	0.81	700
3	0.77	0.80	0.78	700
4	0.89	0.91	0.90	700
5	0.93	0.80	0.86	700
6	0.86	0.90	0.88	700
7	0.88	0.82	0.85	700
8	0.76	0.71	0.73	700
9	0.76	0.82	0.79	700

The CNN itself is predicting the digit 1 better again. It is also getting good results with the digits 0 and 4.

Table 6.10: CNN average results using 100 images

Loss	0.5051
Acc	0.8433
Precision	0.8471
Recall	0.8433
F1	0.8438
Support	7000

The average results show that doubling the amount of data has made the model improve by 8% on average, decreasing the loss too.

CNN with VAE generated images

Table 6.11: CNN VAE using 100 images and a 2D latent space

Digit	**precision**	**recall**	**f1-score**	**support**
0	0.96	0.95	0.95	700
1	0.94	0.98	0.96	700
2	0.82	0.91	0.86	700
3	0.85	0.87	0.86	700
4	0.93	0.92	0.93	700
5	0.81	0.93	0.86	700
6	0.96	0.94	0.95	700
7	0.86	0.87	0.87	700
8	0.88	0.59	0.71	700
9	0.81	0.85	0.83	700

The results of the CNN with VAE have changed, now it is predicting better the digits 0, 1 and 6 instead of the 5 like the last test.

Table 6.12: CNN VAE average results using 100 images and a 2D latent space

Loss	0.9062
Acc	0.8801
Precision	0.8823
Recall	0.8801
F1	0.8776
Support	7000

As for the average results they have improved over the last test by 8% on average. The loss has decrease also.

CNN with Keras data augmentation

Table 6.13: CNN with Keras data augmentation using 100 images

Digit	**precision**	**recall**	**f1-score**	**support**
0	0.95	0.93	0.94	700
1	0.97	0.99	0.98	700
2	0.91	0.97	0.94	700
3	0.94	0.93	0.93	700
4	0.97	0.97	0.97	700
5	0.97	0.99	0.98	700
6	0.98	0.96	0.97	700
7	0.96	0.91	0.93	700
8	0.91	0.91	0.91	700
9	0.94	0.92	0.93	700

Data augmentation with 100 images predicts every digit very well.

Table 6.14: CNN with Keras data augmentation average results using 100 images

Loss	0.2006
Acc	0.9486
Precision	0.9488
Recall	0.9486
F1	0.9485
Support	7000

The average results show that it has improved from the last test by 8% too.

Summary

Table 6.15: Average results of the methods using 100 images and a 2D latent space

	CNN	VAE	AUG
Loss	0.5051	0.9062	0.2006
Acc	0.8433	0.8801	0.9486
Precision	0.8471	0.8823	0.9488
Recall	0.8433	0.8801	0.9486
F1	0.8438	0.8776	0.9485
Support	7000	7000	7000

As every method has improved more or less by 8% from the previous test, the difference between them is maintained. Data augmentation is also over the two other methods and the VAE is in the middle of both, as before.

6.3.3. 250 Images

At the moment, we will be testing with 250 images in total, which is 25 images per digit. The number of epochs that we employ now is 150, as in the Fig. 6.2.3 shows that it is the sweet spot for this amount of data.

Figure 6.3.3 shows how the digits will look like.

Training VAEs and generating new images

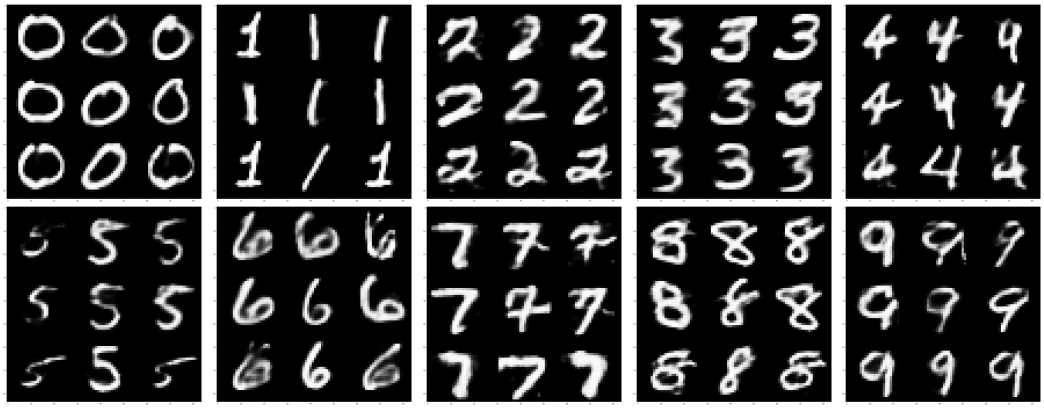


Figure 6.3.3: Generated digits using 250 images and a 2D latent space

Overall they look better than before, but some of them, like the digit 5, are loosing some information.

CNN with no extra images

From now on we will not be looking to the table where it is shown how well the model has learned to predict every digit but only the table where the average results are shown.

So the average results of the CNN trained without extra images using 250 images are as shown in table 6.16.

Table 6.16: CNN average results 250 images

Loss	0.2522
Acc	0.9309
Precision	0.9309
Recall	0.9309
F1	0.9303
Support	7000

Overall the CNN has improved by 9% over the last test though it is more than the double amount of data. We are getting closer to numbers of accuracy where the differences between the methods will be neck and neck.

CNN with VAE generated images

Let us see the average results of the CNN using the generated samples of the VAEs.

Table 6.17: CNN VAE average results using 250 images and a 2D latent space

Loss	0.7026
Acc	0.9011
Precision	0.9071
Recall	0.9011
F1	0.9012
Support	7000

The results have improved by 2% over the last test which is very poor if we take into account that there are more than the double of the amount of data that there was in the

last test. This may be due to the random extract of images from the dataset that has not gotten the most varied samples. This could also mean that is overfitting in excess or even that we have reached the amount of data where the VAE is not helping that much. We will see in the next tests.

CNN with data augmentation

Table 6.18 shows the average results of the CNN with data augmentation of Keras.

Table 6.18: CNN with Keras data augmentation average results using 250 images

Loss	0.1112
Acc	0.9711
Precision	0.9713
Recall	0.9711
F1	0.9712
Support	7000

The results have improved by 3% from the previous test. As the result was already good the improvement is less significant.

Summary

Table 6.19: Average results of the methods using 250 images and a 2D latent space

	CNN	VAE	AUG
Loss	0.2522	0.7026	0.1112
Acc	0.9309	0.9011	0.9711
Precision	0.9309	0.9071	0.9713
Recall	0.9309	0.9011	0.9711
F1	0.9303	0.9012	0.9712
Support	7000	7000	7000

As we can see in table 6.19, the CNN itself is now better than the CNN with the samples of the VAEs. This could be for many reasons so we will keep doing the two tests left. As for the data augmentation is now just 4% above the CNN.

6.3.4. 500 Images

Now we will be testing 500 images in total, which is 50 images per digit.

The digits generated by the VAEs look like this:

Training VAEs and generating new images

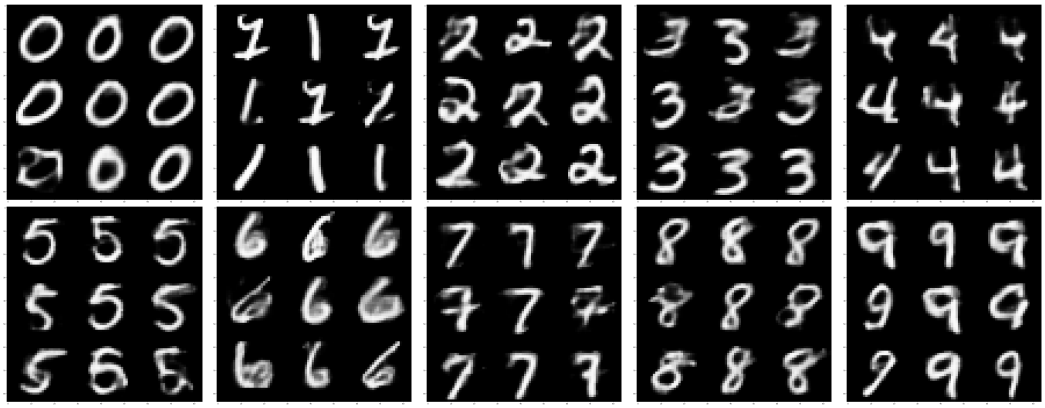


Figure 6.3.4: Generated digits using 500 images and a 2D latent space

CNN with no extra images

The average results of the CNN trained without extra images using 500 images are as shown in table 6.20.

Table 6.20: CNN average results using 500 images

Loss	0.2319
Acc	0.9429
Precision	0.9431
Recall	0.9429
F1	0.9428
Support	7000

The results are improved by 1% overall.

CNN with VAE generated images

Table 6.21: CNN VAE average results using 500 images and a 2D latent space

Loss	0.3275
Acc	0.9424
Precision	0.9437
Recall	0.9424
F1	0.9424
Support	7000

The average result of the CNN with VAE has shown a clearly improve of 4% over the last test where it was worse than the CNN itself.

CNN with Keras data augmentation

Table 6.22: CNN with Keras data augmentation average results using 500 images

Loss	0.0768
Acc	0.9811
Precision	0.9812
Recall	0.9811
F1	0.9811
Support	7000

The improvement of the data augmentation method is of 1% overall.

Summary

Table 6.23: Average results of the methods using 500 images and a 2D latent space

	CNN	VAE	AUG
Loss	0.2319	0.3275	0.0768
Acc	0.9429	0.9424	0.9811
Precision	0.9431	0.9437	0.9812
Recall	0.9429	0.9424	0.9811
F1	0.9428	0.9424	0.9811
Support	7000	7000	7000

Now the results of the CNN and the VAE are neck and neck but the VAE should be at list better than the CNN itself. The data augmentation shows again how well this method works, at least for the MNIST dataset.

6.3.5. 1000 Images

Now we are going to be testing 1000 images in total, which is 100 images per digit.

Training VAEs and generating new images

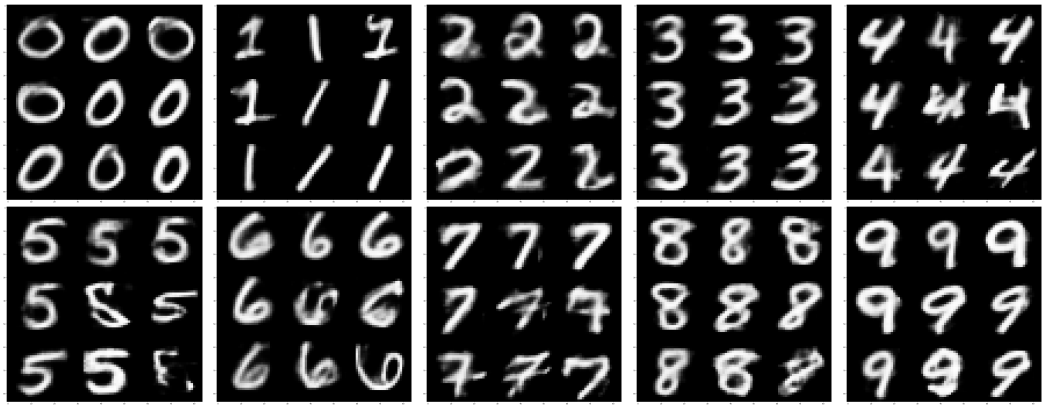


Figure 6.3.5: Generated digits using 1000 images and a 2D latent space

They all look very good except one example of the digit 5, but, overall, great.

CNN with no extra images

The average results of the CNN with no extra images look like this:

Table 6.24: CNN average results using 1000 images

Loss	0.1309
Acc	0.9654
Precision	0.9657
Recall	0.9654
F1	0.9654
Support	7000

It has improved by 2% as it is now a more generous amount of data. Clearly, with this amount of data good results can be obtained without any enhancement.

CNN with VAE generated images

Table 6.25: CNN VAE average results using 1000 images and a 2D latent space

Loss	0.1726
Acc	0.9621
Precision	0.9623
Recall	0.9621
F1	0.9621
Support	7000

The results have improved by 2% from the last test.

CNN with Keras data augmentation

Table 6.26: CNN with Keras data augmentation average results using 1000 images

Loss	0.0529
Acc	0.9836
Precision	0.9837
Recall	0.9836
F1	0.9836
Support	7000

The average results have slightly improved from the last test but just for 0.2%. As the results was already very high the improvement is small.

Summary

Table 6.27: Average results of the methods using 1000 images and a 2D latent space

	CNN	VAE	AUG
Loss	0.1309	0.1726	0.0529
Acc	0.9654	0.9621	0.9836
Precision	0.9657	0.9623	0.9837
Recall	0.9654	0.9621	0.9836
F1	0.9654	0.9621	0.9836
Support	7000	7000	7000

The comparison between CNN and VAE says a lot about what is the problem with the VAE. The generated samples are now introducing a lot of noise to the training and moreover the neural network seems to be overfitting due to the amount of data.

6.3.6. Analysis of the results

The results of the CNN with the generated samples of the Variational Autoencoders are very poor as we are introducing bigger amounts of data. Even with small amounts the data augmentation method is much better.

There could be two problems and two solutions for this:

- One of the problems is that it could be overfitting due to the amount of data and that we are feeding it all of it at the beginning. Data augmentation of Keras augments the data in every epoch and introduces a small amount to the original training set, but the data is not accumulating on every epoch. We could do that on the CNN with the generated images of the VAEs by introducing just a different small amount of the generated images in every epoch.

- The other problem may be the dimensions of the latent space. We are using just 2 dimensions of the latent space which could be translated to a poor reconstruction of the digits, as if it was not learning the most representative features of every digit. We could start by trying 3 dimension on the latent space, for example.

Next we are going to try to improve the proposed method from these conclusions.

6.4. New way of feeding the CNN

We are about to implement the solution of the first problem that we found. The CNN with the new samples that the VAEs had generated was probably overfitting. So we found another way to feed the neural network on every epoch with a different small section of the generated samples that could avoid overfitting.

The structure of this section follows these steps:

- **Implementation:** Subsection where we explain how we implement this method.
- **Results and Comparison:** Subsection where we show the results of the method and compare them with the old one.
- **Analysis of the results**

6.4.1. Implementation

We are going to explain how we have implemented this method. Down below it is shown the modifications done in the code in the training function.

The code:

```
pagination_size = 100
for i in range(0,args.e):
  print("EPOCH %d" % ((i+1)))
  since          = i * pagination_size
  until         = (i+1) * pagination_size

  X_aux = X_vae[since:until]
  Y_aux = Y_vae[since:until]

  X = np.concatenate((X_train, X_aux), axis=0)
  Y = np.concatenate((Y_train, Y_aux), axis=0)

  model.fit(X,Y,epochs=1,
    batch_size=args.b,
    verbose=1,
    validation_data=(X_test, Y_test))
```

The code that we have implemented is using a pagination size of 100, which means

that we will be feeding 100 new generated samples on every epoch without accumulating them. Then we iterate in a loop where the total iterations are the number of epochs and in every epoch we are getting the first 100 images of the generated samples, then the 100 next to them in the next epoch and so on. We concatenate those images with the current training set and fit the model for just 1 epoch doing it every iteration of the loop until complete the number of epochs.

6.4.2. Results and Comparison

Now we show the results of the method that we have just implemented to avoid overfitting. Moreover, we will be comparing it with the old method that we used to train the CNN using the generated samples and with the rest of methods.

Let us start from the very beginning, with 50 images in total.

50 Images

Table 6.28: Comparison of a new way of feeding data to the VAE and the rest of the methods using 50 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.7667	1.8515	0.7899	0.4608
Acc	0.7611	0.8049	0.8499	0.8621
Precision	0.7785	0.8155	0.8526	0.8690
Recall	0.7611	0.8049	0.8499	0.8621
F1	0.7626	0.8037	0.8489	0.8610
Support	7000	7000	7000	7000

In the table above we clearly see the improvement of the new method ("NEW VAE") over the last method ("OLD VAE"), which is of 4-5%. Though it is a lower result comparing it with the data augmentation.

100 Images

Table 6.29: Comparison of a new way of feeding data to the VAE and the rest of the methods using 100 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.5051	0.9062	0.5260	0.2006
Acc	0.8433	0.8801	0.9090	0.9486
Precision	0.8471	0.8823	0.9114	0.9488
Recall	0.8433	0.8801	0.9090	0.9486
F1	0.8438	0.8776	0.9090	0.9485
Support	7000	7000	7000	7000

With 100 images the improvement that we see is less than before being just of 2-3% over the previous method. In this case, data augmentation is further than the last test from our current method in terms of results.

250 Images

Table 6.30: Comparison of a new way of feeding data to the VAE and the rest of the methods using 250 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.2522	0.7026	0.2667	0.1112
Acc	0.9309	0.9011	0.9484	0.9711
Precision	0.9309	0.9071	0.9490	0.9713
Recall	0.9309	0.9011	0.9484	0.9711
F1	0.9303	0.9012	0.9484	0.9712
Support	7000	7000	7000	7000

With 250 images the improvement is remarkable as it is 4-5% better. We have to remind that this is also due to the issues that the old VAE had in this test. Data augmentation is also better.

500 Images

Table 6.31: Comparison of a new way of feeding data to the VAE and the rest of the methods using 500 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.2319	0.3275	0.2768	0.0768
Acc	0.9429	0.9424	0.9566	0.9811
Precision	0.9431	0.9437	0.9566	0.9812
Recall	0.9429	0.9424	0.9566	0.9811
F1	0.9428	0.9424	0.9565	0.9811
Support	7000	7000	7000	7000

With 500 images the improvement is less notable but still worth it. Nevertheless, the data augmentation is still in the top by far.

1000 Images

Table 6.32: Comparison of a new way of feeding data to the VAE and the rest of the methods using 1000 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.1309	0.1726	0.1825	0.0529
Acc	0.9654	0.9621	0.9724	0.9836
Precision	0.9657	0.9623	0.9726	0.9837
Recall	0.9654	0.9621	0.9724	0.9836
F1	0.9654	0.9621	0.9724	0.9836
Support	7000	7000	7000	7000

As we have already said, as we are getting closer to this high percentage the results are getting neck and neck. With 1000 images the new method is close to data augmentation but still far and improving by 1% the old method of the CNN using the generated samples.

6.4.3. Analysis of the results

To conclude this section we have to say that there are noticeable improvements over the last method due to the lack of overfitting that we have achieved thank to this method.

There is still much more room for improvement so let us see how we implement the second solution of the second problem.

6.5. Using 3 dimensions on the latent space

We are about to implement the solution of the second problem. The VAEs that we were training were using just two dimensions on the latent space. The latent space is were the encoding is, the compressed representation of the image, and from that representation the decoder has to reconstruct the image. Two dimensions means that is using that space to keep the main features of the image to then reconstruct it. You could use more than two dimensions to compress the image but as many dimensions you are using the compression is bigger, less compressed.

For this test we are going to utilize 3 dimensions on the latent space and take a look to the results. We do not need to share code because just changing a variable called latent_dim, which is set to 2, to 3 it would do the job.

We show how the digits in the 3 dimensions latent space of the digit 0 would look like for the case of 1000 images, which for the digit 0 is just 100 as is 100 images for each digit:

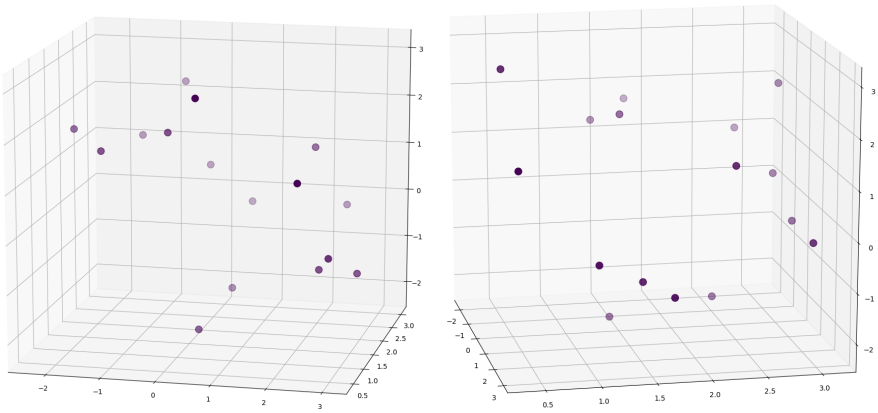


Figure 6.5.1: 3D latent space digit 0

Figure 6.5.2 shows an example of the digits that could be generated with this VAE.

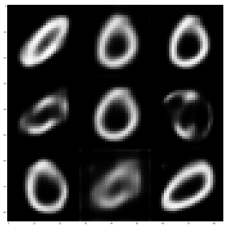


Figure 6.5.2: Generated samples of the digit 0 using a 3D latent space

The digits may or may not look good but it could still be good for the CNN.

6.5.1. Results and Comparison

We are going to run the same tests as before but now the "OLD VAE" will be the last method that we have used which is the one that loaded the images using pagination.

50 Images

Table 6.33: Comparison VAE with 3D latent space and other methods using 50 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.7667	0.7899	0.8169	0.4608
Acc	0.7611	0.8499	0.8549	0.8621
Precision	0.7785	0.8526	0.8588	0.8690
Recall	0.7611	0.8499	0.8549	0.8621
F1	0.7626	0.8489	0.8540	0.8610
Support	7000	7000	7000	7000

100 Images

Table 6.34: Comparison VAE with 3D latent space and other methods using 100 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.5051	0.5260	0.4457	0.2006
Acc	0.8433	0.9090	0.9203	0.9486
Precision	0.8471	0.9114	0.9211	0.9488
Recall	0.8433	0.9090	0.9203	0.9486
F1	0.8438	0.9090	0.9198	0.9485
Support	7000	7000	7000	7000

250 Images

Table 6.35: Comparison VAE with 3D latent space and other methods using 250 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.2522	0.2667	0.2185	0.1112
Acc	0.9309	0.9484	0.9594	0.9711
Precision	0.9309	0.9490	0.9595	0.9713
Recall	0.9309	0.9484	0.9594	0.9711
F1	0.9303	0.9484	0.9593	0.9712
Support	7000	7000	7000	7000

500 Images

Table 6.36: Comparison VAE with 3D latent space and other methods using 500 images

	CNN	OLD VAE	NEW VAE	AUG
Loss	0.2319	0.2768	0.2361	0.0768
Acc	0.9429	0.9566	0.9639	0.9811
Precision	0.9431	0.9566	0.9639	0.9812
Recall	0.9429	0.9566	0.9639	0.9811
F1	0.9428	0.9565	0.9638	0.9811
Support	7000	7000	7000	7000

1000 Images

Table 6.37: Comparison VAE with 3D latent space and other methods using 1000 images

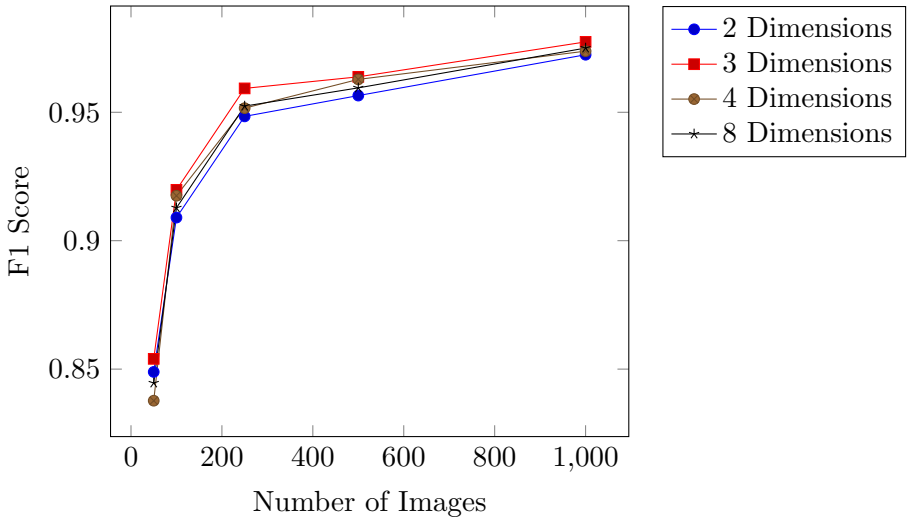
	CNN	OLD VAE	NEW VAE	AUG
Loss	0.1309	0.1825	0.1278	0.0529
Acc	0.9654	0.9724	0.9774	0.9836
Precision	0.9657	0.9726	0.9776	0.9837
Recall	0.9654	0.9724	0.9774	0.9836
F1	0.9654	0.9724	0.9774	0.9836
Support	7000	7000	7000	7000

6.5.2. Using 4 and 8 dimensions on the latent space

In this section we intend to implement the VAEs with 4 and 8 dimensions on the latent space, which means that they should have a larger space where to encode all the data of every image so they can reconstruct them better. As the image input is 28x28(784), if they have a larger space than 2 or 3 dimensions it should mean a better performance, but not necessarily. Let us check that with a couple of test.

In the Fig. 6.5.3 we will see a comparison between the results of the 2, 3, 4 and 8 dimensions on the latent space.

Figure 6.5.3: Graph comparison using F1 score of 2, 3, 4 and 8 dimensions



6.5.3. Analysis of the results

Generally, the new sampled digits using 3 dimensions on the latent space seem to be helping the neural network better than using 2 dimensions on the latent space. The improvement goes from 0.5% up to 2% overall.

We have also tried with more dimensions. We have done a test for 4 dimensions and another one for 8 dimensions. As the Fig. 6.5.3 shows, more dimensions on the latent space are no longer helping to improve the results.

After modifying the latent spaces there could be another test we can do. Data augmentation is known to be well implemented for this dataset, the MNIST dataset, so that is why it is doing that well overall. We could try to put together the generated samples of the VAEs working on the 3 dimensions of the latent space and the data augmentation of Keras to see if there is an improvement over the data augmentation itself.

6.6. Variational Autoencoders with Keras data augmentation

Finally, we put together the method that we use with the Variational Autoencoders to train the CNN and the Data Augmentation method that is used to train the CNN.

6.6.1. Implementation

To do this, we unite both codes as follows:

```
datagen = ImageDataGenerator(
  rotation_range=20,
  width_shift_range=0.2,
  height_shift_range=0.2,
  horizontal_flip=False)

pagination_size = 50
for i in range(0,args.e):
  print("EPOCH %d" % ((i+1)))
  since          = i * pagination_size
  until         = (i+1) * pagination_size

  X_aux = X_vae[since:until]
  Y_aux = Y_vae[since:until]
```

```
X = np.concatenate((X_train, X_aux), axis=0)
Y = np.concatenate((Y_train, Y_aux), axis=0)

model.fit_generator(datagen.flow(X_train, Y_train, batch_size=args.b),
  steps_per_epoch=len(X_train), validation_data=(X_test, Y_test),
  epochs=1,
  verbose=1)
```

We declare the ImageDataGenerator with the same parameters as we did with the data augmentation before. We also declare the pagination size but this time with a size of 50 because we will have too much data instead. The process that we follow is the same as the new one with the Variational Autoencoders but instead of fitting the model as normally we fit it with fit_generator as it is the way that is fitted in data augmentation.

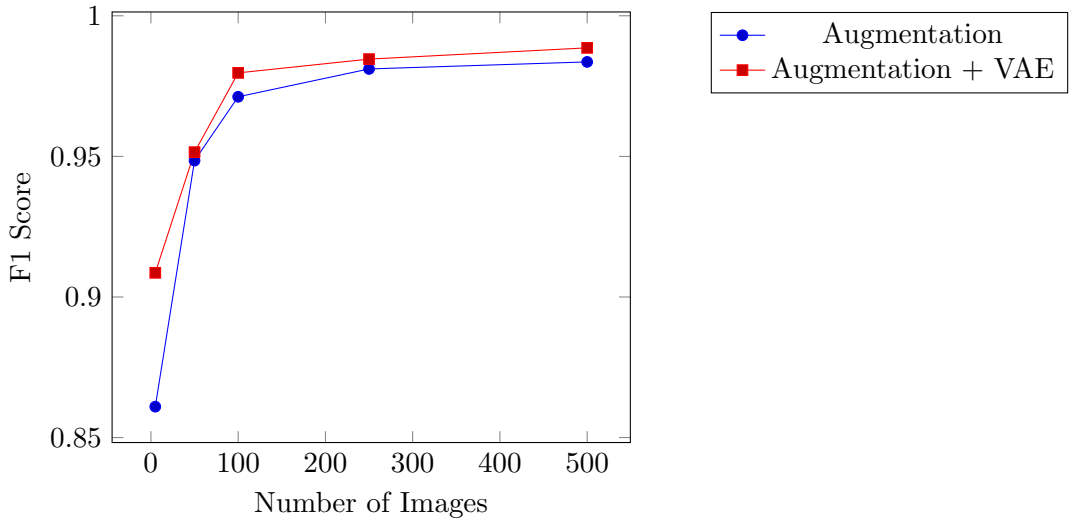
6.6.2. Results and Comparison

We are about to compare the results of this experiment with the ones of the data augmentation that we already ran. We compare them using the F1 Score which has given us very fair and consistent results.

Table 6.38: Comparison using F1 score of Keras data augmentation and Keras data augmentation with VAE

	Augmentation	Augmentation + VAE
50 Images	0.8610	0.9086
100 Images	0.9485	0.9515
250 Images	0.9712	0.9797
500 Images	0.9811	0.9846
1000 Images	0.9836	0.9886

Figure 6.6.1: Graph comparison using F1 score augmentation and augmentation with VAE



6.6.3. Analysis of the results

As we can see in table 6.38 and in Fig. 6.6.1 the union of Data Augmentation and Variational Autoencoders improves the original Data Augmentation, overall.

These results are very remarkable because thank to this method, using just 5 images per digit (50 in total), 90% of success is achieved, for which the network alone would need about 200 images and about 100 if we use only the data augmentation of Keras.

6.7. Summary Results

To conclude the experiments we are going to summarize all the results that we have obtained in this section.

Table 6.39: Summary of experiments

Latent Space	Training Size	CNN	VAE	AUG	AUG + VAE
2	50	0.7626	0.8489	0.8610	-
	100	0.8438	0.9090	0.9485	-
	250	0.9303	0.9484	0.9712	-
	500	0.9428	0.9565	0.9811	-
	1000	0.9654	0.9724	0.9836	-
3	50	0.7626	0.8540	0.8610	0.9086
	100	0.8438	0.9198	0.9485	0.9515
	250	0.9303	0.9593	0.9712	0.9797
	500	0.9428	0.9638	0.9811	0.9846
	1000	0.9654	0.9774	0.9836	0.9886
4	50	0.7626	0.8377	0.8610	-
	100	0.8438	0.9175	0.9485	-
	250	0.9303	0.9516	0.9712	-
	500	0.9428	0.9628	0.9811	-
	1000	0.9654	0.9738	0.9836	-
8	50	0.7626	0.8446	0.8610	-
	100	0.8438	0.9128	0.9485	-
	250	0.9303	0.9524	0.9712	-
	500	0.9428	0.9595	0.9811	-
	1000	0.9654	0.9750	0.9836	-

7 Conclusion

7.1. Project Summary

The purpose of this project is to accomplish a new method to overcome the lack of data. In the literature the strategy that is accustomed to achieve this task is data augmentation which is a method that artificially creates new data based on the modifications of the existing data. The heuristics underlying this modifications are very dependent on which processes are suitable for the classification task at issue. In this project we introduce an alternative using Variational Autoencoders which are powerful generative models. These are capable of extracting latent values from input variables to generate new information without the user having to take specific decisions. We explain and implement them from scratch, and try to make the most out of them. The results that we obtain are compared with current methods side by side. After the comparisons, we make some conclusions and give some final thoughts.

7.2. Evaluation

The results that we obtained in the experimentation section were very consistent and promising. Variational Autoencoders themselves could not improve data augmentation at the end. At the beginning of the experiments the results were improving the neural network process that had not enhancements but were far behind the ones of the data augmentation of Keras. This is already an improvement, since this method learns the transformations, while in augmented of traditional data the programmer has to set the types of transformations to be carried out since all the transformations for all types of data are not always good (for example, it does not make sense to invert digits or letters but to rotate them a bit).

However, with many improvements over Variational Autoencoders we could achieve close results to the current best method that is data augmentation. Moreover, we tried to put it along with our method to see if this helped improve the results. And in fact, it improved it by feeding the original data along with the generated samples of the Variational Autoencoders to the data augmentation method.

Data augmentation is well known to be a very good solution for some recognized datasets. As for the MNIST dataset that we have used, it works quite well. Nevertheless, Variational Autoencoders promise to be a more general solution as it could work with any type of data.

7.3. Further Work

As we have already talked about, we have stuck to the MNIST dataset to compare the results of the different methods. Considering Variational Autoencoders work for any kind of data we can not tell the future work we could extract from this project, but we are going to number some of it:

- **New datasets:** It is unfortunate that we could not try some other datasets where data augmentation were not that good for that specific dataset. So some further work could be to try new datasets the same way that we have done that with the MNIST dataset.

- **Experiment with latent dimension:** The latent dimension of the Variational Autoencoders could be used to compress data or to generate whatever type of new data, as for instance it is done in this report [6].

- **Alter existing data:** We have used them to generate new samples from already known ones with some variation in them. But another purpose of VAEs is to alter existing data. You could, for instance, add new pair of glasses to a person that in the beginning does not have them.

- **Create:** With Variational Autoencoders you could create new master pieces of art like drawings, new 3D models of any kind and even create music. The possibilities are endless.

7.4. Final Thoughts

I was a novice of neural networks when I started this project. I had done just a few projects of Machine Learning so I had a huge room for improvement. When I started this project I knew it was going to be difficult but that made it more fascinating. I could not have done this project without the help of my tutors who have been there for any question I had. I have definitely learned very much from this project, I have a very deeper understanding of neural network at this moment thank to it. Nevertheless, there is a long way to go to get a very good comprehension of Machine Learning, I am very

excited to explore it in depth.

I am very grateful that this project accomplish the objectives that it was meant to achieve. In some way, with this research we have contributed to the community and to the world, also opening a door to any improvement or complement to this field that any researcher want to follow.

Bibliography

[1] Adadelta: An adaptive learning rate method. urlhttps://arxiv.org/abs/1212.5701.

[2] Autoencoders, minimum description length, and helmholtz free energy. https://www.cs.toronto.edu/~hinton/absps/cvq.pdf.

[3] Building autoencoders in keras. https://blog.keras.io/building-autoencoders-in-keras.html.

[4] Data augmentation techniques in cnn using tensorflow. https://medium.com/ymedialabs-innovation/data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9.

[5] The effectiveness of data augmentation in image classification using deep learning. https://arxiv.org/pdf/1712.04621.pdf.

[6] Extracting a biologically relevant latent space from cancer transcriptomes with variational autoencoders. urlhttps://www.ncbi.nlm.nih.gov/pubmed/29218871.

[7] Image augmentation for deep learning with keras. https://machinelearningmastery.com/image-augmentation-deep-learning-keras/.

[8] Imagedatagenerator class and its methods. https://keras.io/preprocessing/image/.

[9] An introduction to artificial intelligence. https://hackernoon.com/understanding-understanding-an-intro-to-artificial-intelligence-be76c5ec4d2e.

[10] Intuitively understanding variational autoencoders. https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.

[11] Learning internal representations by error propogation. https://dl.acm.org/citation.cfm?id=104293.

[12] Nanonets : How to use deep learning when you

have limited data. https://medium.com/nanonets/nanonets-how-to-use-deep-learning-when-you-have-limited-data-f68c0b512cab.

[13] Neural networks and principal component analysis: Learning from examples without local minima. https://www.researchgate.net/publication/222438485.

[14] Nist special database 19 handprinted forms and characters database. https://www.nist.gov/sites/default/files/documents/srd/nistsd19.pdf.